CORBA-based Common Software for the ALMA project

G. Chiozzi[*a], B. Gustafsson[a], B. Jeram[a], M. Plesko[b], M. Sekoranja[b], G. Tkacik[b], K.Zagar[b]
[a] European Southern Observatory; [b] Jozef Stefan Institute

ABSTRACT

The Atacama Large Millimeter Array (ALMA) is a joint project between astronomical organizations in Europe and North America. ALMA will consist of at least 64 12-meter antennas operating in the millimeter and sub-millimeter range, with baselines up to 14 km. It will be located at an altitude above 5000m in the Chilean Atacama desert.

The ALMA Common Software (ACS) provides a software infrastructure common to all partners and consists of a documented collection of common patterns and of components that implement those patterns.

The heart of ACS is an object model based on Distributed Objects (DOs), implemented as CORBA objects. The teams responsible for the control system development use DOs as the basis for components and devices such as an antenna mount control.

ACS provides common CORBA-based services such as logging, error and alarm management, configuration database and lifecycle management. A code generator creates a Java Bean for each DO. Programmers can write Java client applications by connecting those Beans with data-manipulation and visualization Beans.

ACS is based on the experience gained in the astronomical and particle accelerator domains, and reuses and extends proven concepts and components. Although designed for ALMA, ACS can be used in other new control systems, since it implements proven design patterns using state of the art, stable and reliable technology.

This paper presents the architecture of ACS and its status, detailing the object model and major services.

Keywords: CORBA, Control Software, ALMA

1. INTRODUCTION

Since its beginning, the ALMA project [1] has been characterized by complexity due to the wide geographical distribution of its development teams and their diverse development cultures. The number of applications and developers will increase to very large numbers.

To alleviate these problems, we have introduced a central object oriented framework, the ALMA Common Software (ACS)[5]. It is located in between the ALMA application software and other basic commercial or shared software on top of the operating systems. It provides a well-tested platform that embeds standard design patterns and avoids duplication of effort. At the same time it is a natural platform where upgrades can be incorporated and brought to all developers. It also allows, through the use of well-known standard constructs and components, other team members who are not authors of ACS to easily understand the architecture of software modules, making maintenance affordable even on a very large project.

In order to avoid starting from scratch, we have evaluated emerging systems that could provide a good basis for ACS and would bring to the project CORBA[12] and other new technological know-how. We then began a fruitful collaboration between ESO and JSI that, through exchange of experience and ideas from our previous projects has brought us to the concepts and implementation of ACS. For more details on the considerations that have led to the concepts behind ACS see [2], [6] and [7].

During the first phases of the project, we have concentrated on the aspects related to the Control System for ALMA, driven by the requirements of the ALMA Test Interferometer[4]. Now the ACS team is working with the ALMA Architecture team to seamlessly extend ACS to cover the needs of the higher-level software sub-systems[3]. The objective is to use uniformly across the whole project, whenever possible, the same basic concepts, models and tools.

[*] gchiozzi@eso.org; phone: +49-89-32006 543; WWW: http://www.eso.org/~gchiozzi; European Southern Observatory, Karl-Schwarzschildstrasse, 2, D-85748, Garching bei Muenchen, Germany.

2. ACS ARCHITECTURE

ACS is based on the object oriented CORBA middleware, which gives the infrastructure for the exchange of messages between distributed objects and system wide services [8]. Whenever possible, ACS features are implemented using off-the-shelf components; ACS itself provides in this case the packaging and the glue between these components. At the same time, whenever convenient ACS hides all details of the underlying mechanisms, which use many complex features of CORBA such as queuing, asynchronous communication, thread pooling, life-cycle management, etc.

2.1 Component-Container model

The ACS Architecture is founded on the Component-Container model [17].

As illustrated in Fig.1, a client (which could itself be a Component residing in another Container) accesses the services of Components via their published service interfaces. The interfaces published by Components 1 and 2 are shown inside the Container, because in these two cases, access to their services is mediated by the container itself, which may add additional services such as client authorization verification to those provided by the component. Here, the Container is acting as a "tight" container. For Component 3, on the other hand, the container is "porous"; access to the component is direct, although the container will be used to supply the client with the initial reference to Component 3. This might be desirable when performance considerations outweigh the benefits that a tight Container could provide in the way of added services. These two modes, "*tight*" and "*porous*", reflect the expected use of this model in the data flow and control system areas, respectively. The client, however, receives only a reference to its needed component, and does not know which of these two container types it is dealing with. A higher-level Manager coordinates the Containers to provide Component life cycle management and location services.

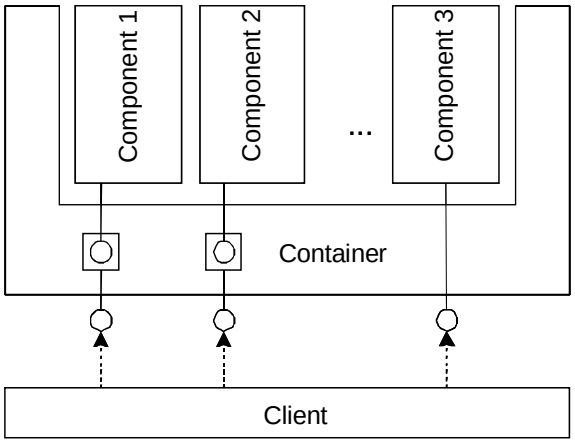

Fig . 1: Component/Container model

Containers provide an environment for Components to run in, with support for basic services like logging system, configuration database, persistency and security. Developers of Components can focus their work on the domain-specific "functional" concerns without having to worry about the "technical" concerns that arise from the computing environment in which their components run.

The division of responsibilities between components and containers enables decisions about where and when individual components are deployed to be deferred until runtime, at which point configuration information is read by the container. If the container manages component security as well, authorization policies can be configured at run time in the same way.

Commercial implementations of the Component-Container model are quite popular in industry at present, with Sun's Enterprise Java Beans and Microsoft's .NET being the prime examples. A vendor-independent specification, the Corba Component Model (CCM), is under development, but it is not complete, and production implementations do not yet exist. These are rather comprehensive systems, and require a wholesale commitment from developers to use the languages and tools supplied. For our application domain we do not need all the power offered by such systems and ACS implements an infrastructure that is more lightweight. More details on the Component/Container implementation will be provided in section 5.

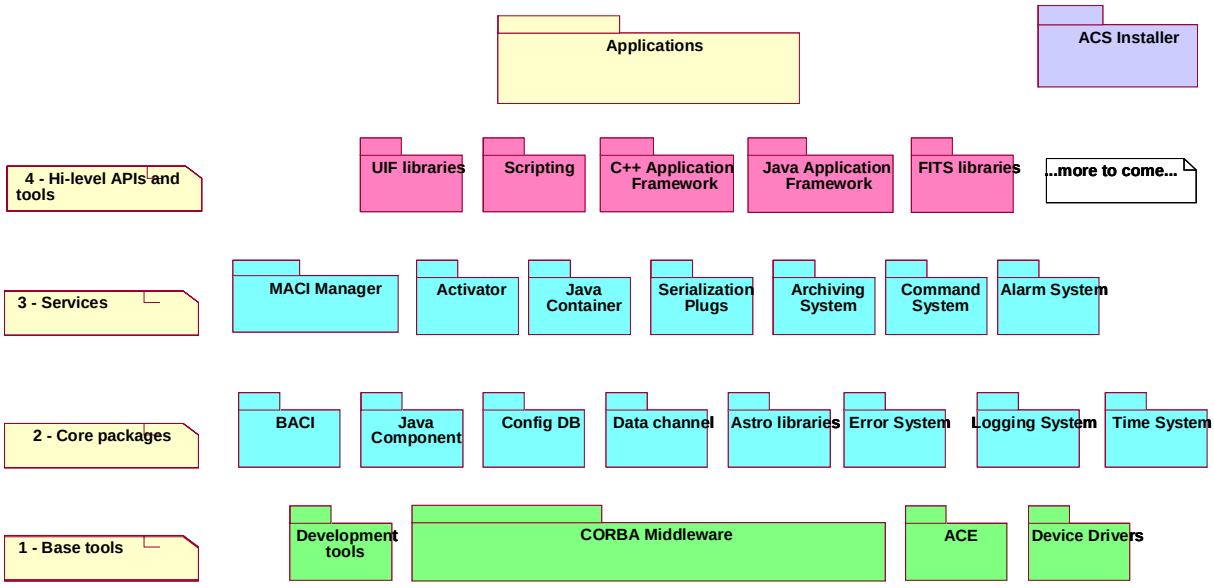

Fig . 2: ACS Packages

2.2 ACS Packages

The UML Package Diagram in Fig.2 shows the main packages in which ACS has been subdivided. For more details, refer to the ACS Architecture, available on the ACS Web Page[2].

ACS has a classical layered architecture, where packages are allowed to use services provided by other packages on lower layers and on the same layer, but not on higher layers.

Each package provides a basic set of services and tools that shall be used by all ALMA applications.

2.2.1 Base Tools

The bottom layer contains base tools that are distributed as part of ACS to provide a uniform development and run time environment on top of the operating system for all higher layers and applications. These are essentially off-the-shelf components and ACS itself simply provides packaging, installation and distribution support. This ensures that all installations of ACS (development and run-time) will have the same basic set of tools.

2.2.2 Core Packages

This second layer ensures standard interface patterns and implements essential services, necessary for the development of any application. Among these:

- **Basic Access Control Interface (BACI)**
  We expect that the ALMA control system will be mainly implemented in C++, to satisfy performance and real time requirements. This package provides the implementation of C++ Components, introducing the concepts of *Distributed Object*, *Property and Characteristic* [9]. The object paradigm of CORBA is fully applied: each entity in the control system is defined as a Component type and is represented by one specific CORBA interface that subclasses the base *Distributed Object (DO)*. Each DO is further composed of *Properties* that correspond to what are called controlled points, channels or tags in Supervisory Control and Data Acquisition systems (SCADA). Each *Property* is an object too, described by *Characteristics* such as min/max values or units. A more detailed description is given in section 3.

- **Java Component**
  Higher-level software, in particular data-flow applications, will be instead mainly developed in Java. This package provides a Java implementation for the Component model. Java Components will not enforce the *Distributed Object, Property and Characteristic* paradigm, which is particularly suited for control system components, but will be otherwise similar and interoperable with Distributed Objects. Furthermore, it will be possible to have serializable and persistent Java Components. The Java Component package is currently under design.
- **Configuration Database**
  This package addresses the problems related to defining, accessing and maintaining the configuration of a run-time system. For each Component in the system, there are configuration parameters that must be configured in a persistent store and read when the Component is started up or re-initialized. This package is described in section 4.
- **Data Channel**
  The Data Channel provides a generic mechanism to asynchronously pass information between data publishers and data subscribers, in a many-to-many relation scheme. It is based on the CORBA Notification Service [12].
- **Error System**
  API for handling and logging run-time errors, tools for defining error conditions; tools for browsing and analyzing run-time errors.
- **Logging System**
  API for logging data, actions and events. Transport of logs from the producer to the central archive. Tools for browsing logs. It is based on the CORBA Telecom Logging Service [12].
- **Time System**
  Time and synchronization services.

2.2.3 Services

The third layer implements higher-level services. Among these:

- **Management and access control interface (MACI)**
  Design patterns, protocols and meta-services for centralizing access to ACS services and Components, to manage the full life cycle of Components, including persistence, and to supervise the state of the system [10]. This has been split in three packages:
  - **MACI Manager** contains the higher-level system supervisor (Manager). See section 5.
  - **Activator** implements the C++ Container. See section 5.
  - **Java Container** implements the Java Container for higher-level application components. For Java applications we think that a "*tight Container*"(see section 2.1) model is more appropriate. It will also include significant support for serialization of entity data objects. The Java Container is currently under design.
- **Archiving System**
  API tools and services for archiving and monitoring data and events.

2.2.4 Application Frameworks and High-level APIs

The fourth and last layer provides higher-level APIs and tools. The main goal of these packages is to offer a clear path for the implementation of applications, in order to obtain implicit conformity to design standards. Among these, we mention:

- **UIF Libraries**
  Development tools and widget libraries for User Interface development. Java user interfaces are based on the ABeans library that wraps CORBA objects within Java Beans, which are then connected with commercial data-manipulation and visualization Beans using visual tools or programmatically[11]. See section 6.
- **ACS C++ and Java Application Frameworks**
  Implementation of design patterns and to allow the development of standard applications.

3. BACI

BACI is a control system framework that uses and extends component-based, distributed computing and object-oriented concepts in order to standardize software, shorten its development cycle and increase its reliability. The heart of BACI is a distributed object model (see BACI Class Diagram).

All common telescope components such as antenna mount, antenna control unit, correlator, etc. are Components defined by means of *Distributed Objects (DOs). DOs* are implemented as CORBA objects that are remotely accessible from any computer through the client-server paradigm.

Each *DO* is further composed of *Properties.* A *DO* can also contain references to other *DOs* to build hierarchical structures of components.

Both *DOs* and *Properties* have specific *Characteristics*, e.g. a Property has a minimum, a maximum, units…. The common behavior of *DO* and *Property* has been factorized in the *Named Component* common base class. The methods of a *Named Component* allow retrieval of these *Characteristics.* Notice the programatically characteristics are read-only. For example, all constants related to a *Property* such as min/max, and description are obtained by clients directly from the *Property* by means of remote methods - no direct access to the configuration database is necessary.

Values of the *Properties* are updated asynchronously by means of monitor objects.

While there are in principle an infinite number of *DO* types, for example one for each physical controlled device, there are very few different *Property* types: in principle one for each primitive data type and one for each sequence of primitive data types. [9].

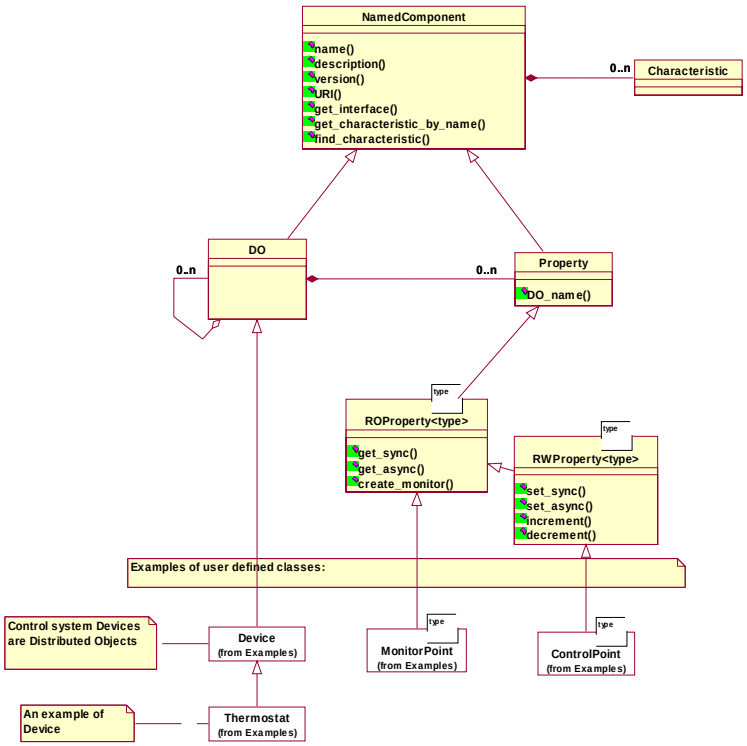

Fig . 3: BACI Class Diagram

From a conceptual point of view, BACI is a definition of interface patterns with the use of IDL, not a definition of an application programming interface (API). BACI prescribes a model for *Distributed Objects* that is commonly used in control systems. BACI defines patterns that are the largest common denominator that can be found among different types of experimental facilities. As such it should enable portability of core control software. Using the patterns and libraries of BACI one can create and implement the API for a given control system in three steps:

1. The creator of the control system must first define interfaces for *Properties*, according to the rules set forth in BACI. This could be done by a code generator, which generates IDL, implementation code and database structures from the same input.
2. Once the different types of *Properties* have been defined and the IDL for them exists, the experts for the individual devices describe each device as a set of the existing *Properties*. Only the interfaces are described, again using IDL.

3. Programmers of concrete device behavior use the IDL components to define their devices and use the generic library to write servers. All the common tasks of servers, such as handling and managing client requests, CORBA Object (COB) life-cycle management, dispatching monitor callbacks and initializing servant code is done by the generic libraries that are provided by ACS in the BACI and MACI (Management and Access Control Interface) packages(see section 5).

All the IDL interfaces that have been defined for the *Distributed Objects* are the actual interface to the specific control system under development. For example, the interface for a simple Lamp would look like:

```
interface Lamp: DistributedObject {
   void on();
   void off();
   readonly attribute RWdouble brightness;
};
```

where `brightness` is a `read-write` property of type `double` ("`readonly attribute`" is the IDL syntax to instantiate a data member, in this case of the complex type `RWdouble`).

As there are many distinct interface types and each method has a specific signature, this is considered a *"wide"* interface in contrast to the *"narrow"* one, which is common in many control systems:

```
ctrlObj.remote(device,"msg",dataIn,dataOut)
```

where the object named ctrlObj keeps a few methods that encapsulate all communication with the server. Here, device represents a generic device and the parameters dataIn and dataOut are some generic containers that keep any type of data. The string "msg" is a command that the device server understands – unless the programmer has mistyped it. Based on the meaning of the command string, the device server interprets the input data from the dataIn container and packs data into the dataOut container.

A significant advantage of the wide interface is, that the explicit method call on objects is safer, as many errors are discovered automatically at compile-time by the compiler. Also, the IDL is itself the *DO* manual. A simple look at the object definitions provides all needed insight into the *DO* commands. Adding it all up and considering the advantages of object-oriented concepts such as inheritance, we see that the wide interface is natural for CORBA: all *DOs* are arranged by interface type. Each device interface inherits from a basic *Distributed Object*, which defines basic data such as name, position and security/access.

Advanced CORBA features and services further empower the concept of the wide interface. Cobra's reflection or introspection allows run-time discovery of interfaces and methods. It is possible to find all interfaces that are supported by running *DOs* in the system. It is equally straightforward to find first all devices of a given interface, and then find all methods of a given device. The clean structure of *DOs* makes the interpretation easy. ACS includes an application that uses all the above-mentioned features and allows the operator to invoke any command on any *DO* through a series of lists and menus. This application, called Object Explorer, is made completely generic through use of CORBA's dynamic invocation interface. Any future *DO* that is modeled according to the BACI model will be discovered and controlled the moment it becomes available to the system.

4. CONFIGURATION DATABASE

The ACS Configuration Database addresses the problems related to defining, accessing and maintaining the configuration of a system based on ACS. For each Distributed Object in the system, there is a set of configuration parameters that have to be configured in a persistent store and read when the DO is started up or re-initialized. This includes the *structure* of the system, i.e. which DOs are part of the system and their inter-relationships.

There are 4 different issues related to this problem:

1. input of data by the user
   System configurators define the structure of the system and enter the configuration data.

2. storage of the data
   The configuration data is kept in a database.
3. maintenance and management of the data (e.g. versioning)
   Configuration data changes because the system structure and/or the implementation of the system's components changes with time and has to be maintained under configuration control.
4. loading data into the ACS Components
   At run-time, the data has to be retrieved and used to initialize and configure the DOs.

4.1 Three-tier database-access architecture

The architecture of the ACS configuration database is based on three layers[16]:

1. The Database Itself
   It is the database engine used to store and retrieve data. It may consist of a set of XML files in a hierarchical file system structure or it may be a relational database or another application specific database engine.
2. The Database Access Layer (DAL)
   hides the actual database implementation from applications, so that the same interfaces are used to access different database engines. For each database engine a specific DAL CORBA Service is implemented. The DAL is defined in terms of CORBA IDL interfaces and applications access data in the form of XML records or CORBA Property Sets.
3. The Database Clients access data from the database using only the interfaces provided by the DAL.
   Data Clients, like Activators, Managers and DOs retrieve their configuration information from the Database using a simple read-only interface. On the other hand, CDB Administration applications are used to configure, maintain and load data in the database using other read-write interfaces provided by the DAL layer.

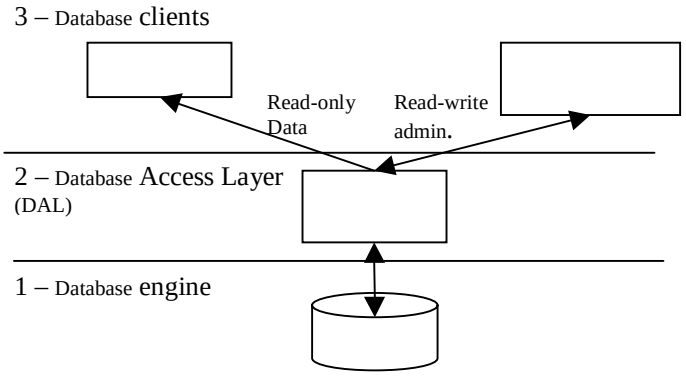

Fig . 4: Configuration Database architecture

4.2 CDB Modeling and Population

Simplicity of administration of the Configuration Database is extremely important for the final running system. Often, the user is not the programmer of the DO and has no knowledge of CORBA, maybe not even of ACS. Furthermore, the configuration data changes often, in particular during construction and commissioning. There is also need for a tool that displays the schema of the data and the relations among different data items.

The architecture of the Data Administration part consists therefore of modules providing different levels of abstraction.

At a basic level a Database Loader application is used to manipulate database description files in an XML format independently of the underlying database engine and load them into the Database using the DAL administrator interfaces. The Database Loader also takes care of validating the data via XML schemas.

On top of the Database Loader comes a Visual Configuration Tool (CT). The CT allows visual editing and validation of the structure/schema of the configuration database and filling in the values inside the instantiated database. It allows switching between different views: source, members, hierarchy, editions, and visual composition. An existing CDB can be parsed and displayed visually, for re-engineering.

At the highest level is a modeling tool. This tool should be used by a System Architect not just to describe and fill in the Configuration Database, but also to define the structure of the system itself.

Using such a tool, the following are specified:

- the DOs available in the system
- the Properties that compose the DOs
- the Characteristics that describe the system
- relationships between the DOs, such as:
    - inheritance (a DO class being a special case of another DO)
    - containment by reference(a DO needs another DO to perform its job)
    - operations that can be performed on DOs

The output of the modeling tool (that would be based on a commercial UML modeler) is a UML representation of the system. This output is intended to be used by code generators to produce the schema of the configuration database and to fill it in, but also to produce, for example, the IDL interfaces, the actual servant code and Java Beans for developing Java clients using visual composition tools.

4.2.1 Switching between instances of Configuration Databases

If all configuration data are stored in a central database, then this database must be up and running all the time during the development of servants and during the time configuration data is entered. Given that ALMA is developed in different timescales in different places all around the world, this is not a viable solution. For test purposes there will be many small CDB "fragments".

In order to allow switching between different instances of the Configuration Database, the reference to the DAL used by each DO is provided as a common service by the Activator/Container inside which the DO lives. This means that each Container has the freedom to use a central CDB or a specific one for development, maintenance or testing purposes.

5. MACI

Management and Access Control Interface (MACI) implements the Container part of the Component-Container model. It is a service that knows about all the Distributed Objects that together compose the control system and manages their interconnections and lifecycle. MACI has two major elements:

- *Activators* are the C++ Containers. They are deployed locally on all hosts involved in the control system, ranging from real-time local control units to high-performance workstations. Their primary task is preparing the local environment in which DOs (the Components) are created, giving them all the resources they need to perform their tasks, such as CORBA connectivity, connection establishment with other DOs and Configuration Database access.

- The *Manager*, which is set up at one central location that is widely known across the entire system. The Manager is acquainted with all the Distributed Objects and

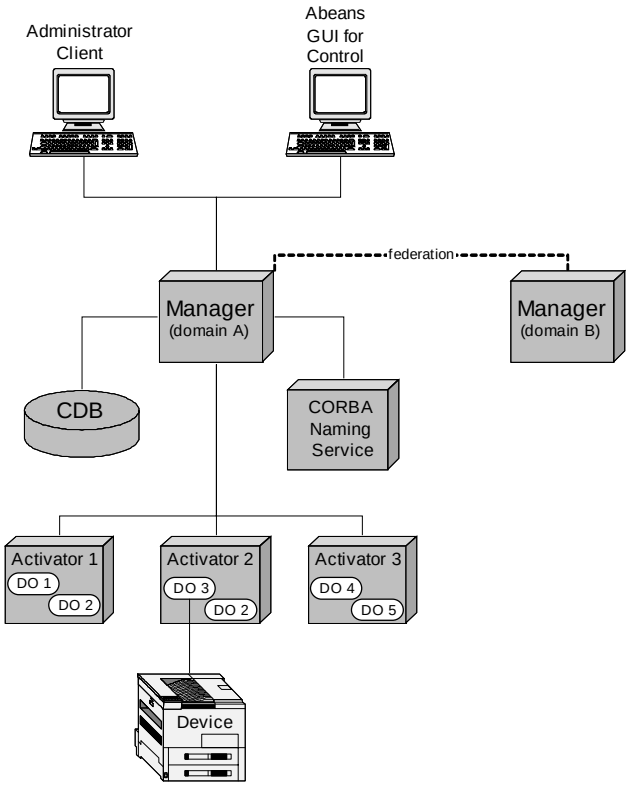

Fig . 5: MACI deployment diagram.

Activators in the system, as well as other resources, such as configuration database and CORBA services. In particular, the Manager closely cooperates with the CORBA Naming Service, in which it publishes all of its acquaintances, making them accessible to non-MACI-aware CORBA software.

Within the scope of MACI, a Component is the Distributed Object (DO). It is given a unique, non-volatile identification, called Component Unique Resource Locator (CURL). CURL does not directly specify the host on which the DO resides: instead, it serves as a handle through which the designated DO can be accessed.

We are currently designing the Java Containers as the counterpart of Activators and Java Components as the counterpart of DOs for code to be written in Java. Java Containers and Components will also be administrated and coordinated by the MACI Manager.

5.1 Control System Lifecycle

After a host computer in the control system has been brought on-line it starts its Activator. The Activator establishes a connection to the Manager via CORBA and introduces itself to the system. If the Manager cannot be found, either because it has not been started yet, or because of a transient networking failure, the Activator attempts to find the Manager periodically.

The Manager checks to see if the Activator that just connected with it must start hosting any Distributed Objects at its startup. If so, it orders the Activator to bring these DOs to life in a process called *activation*.

Once started, the operators may use the control system, which they do using GUI tools typically written using ABeans (see section 6). Such a GUI tool first connects to the Manager, asking it for any Distributed Objects it needs to work with. If the requested Distributed Object does not exist yet, the Manager attempts to activate it ad-hoc.

Shutting down all the Activators shuts down the control system. In the process, all Distributed Objects are destroyed, giving them a chance to properly release the resources they are using.

5.2 Distributed Object Lifecycle

A Distributed Object's lifecycle begins when the Activator activates it. Activation requires the Manager to supply at least three parameters to the Activator: the type of the Distributed Object to create (a class in the object-oriented programming jargon), the location of the DO's program code and the unique instance identification of the DO, which coincides with the CURL.

During activation, the Activator locates the DO's program code on a medium available to the Activator's host and loads it dynamically.

Once the program code is dynamically loaded, it is requested to construct the Distributed Object. The DO is given a unique name (the CURL), and the ability to communicate with the Activator. Typically, the DO would ask the Activator to retrieve its configuration parameters form the CDB, again using its CURL as the identification of the configuration data to be retrieved. Based on the configuration data, the DO would initialize its internal state.

The Activator binds the initialized DO with CORBA's Portable Object Adapter (POA), thus giving it an Interoperable Object Reference (IOR) and allowing it to respond to CORBA invocations. The Manager associates the CURL and the IOR of the newly created object, and asks it to perform any post-initialization.

The Distributed Object's lifecycle is terminated when the Manager decides the object is no longer needed, for example, when the object is not used by anyone in the system and its presence is not crucial. The Manager instructs the hosting Activator to destroy the object, which it does by calling the object's destructor, freeing all memory and other resources associated with the object, disconnecting the object from the POA, and unloading the object's program code from memory. The process of terminating the DO is called *deactivation*.

5.3 Other MACI Features

Since all Distributed Objects are requested through the Manager, the Manager is in a unique position to handle:

- **Garbage Collection:** Keep track of the number of clients of a given DO, as well as the identities of these clients. Once the number of clients for a given DO falls to zero, and the DO is not marked as *immortal*, the DO is deactivated.
- **Authorization:** Allow or deny the client a permission to access a given DO, based on the authorization settings associated with the client's identity.
- **Fault Tolerance:** Detect failures of computers hosting the DOs, and relocate the DOs to a safer place, provided that a DO is not tied to the hardware or other resources of its host.
- **Load Balancing:** Stateless CPU intensive Distributed Objects that are not tied to machine-specific resources can be cloned across several Activators. When requested via the Manager, the Manager can return a reference to a different instance of the object, thus distributing processing load equally across several computers.
- **Federation:** In complex systems, there can be several Managers, each one in charge of its own *domain*, where the coupling between domains is very weak. Inter-domain communication is still possible so that clients in one domain can request DOs in another, since the domain can be inferred from the CURL. This complexity is handled by the Manager through federation.

Not all of these features are already implemented because there is no immediate need for them in the ACS, but should a need appear, they could be added without requiring any changes to existing implementations of Distributed Objects and GUI tools.

6. ABEANS

When writing a client or GUI application the programmer is most often confronted with numerous problems in the areas of error handling, timeout handling, logging, communication system details, resource initialization and destruction and the like.

On the client level, we need a solid framework of components that can be directly assembled into powerful applications. We have developed such a framework in Java using its component model, called Java Beans: The ACS distributed objects are wrapped into specially developed Java Beans called Abeans, which provide a rich set of tools that clients frequently need. This allows applications to be written easily, even by non experienced Java programmers.

A Java Bean is a reusable component that can be manipulated with readily commercially available visual editing tools. We use standard, commercial and also homemade beans, when the commercial ones don't provide the necessary functionality.

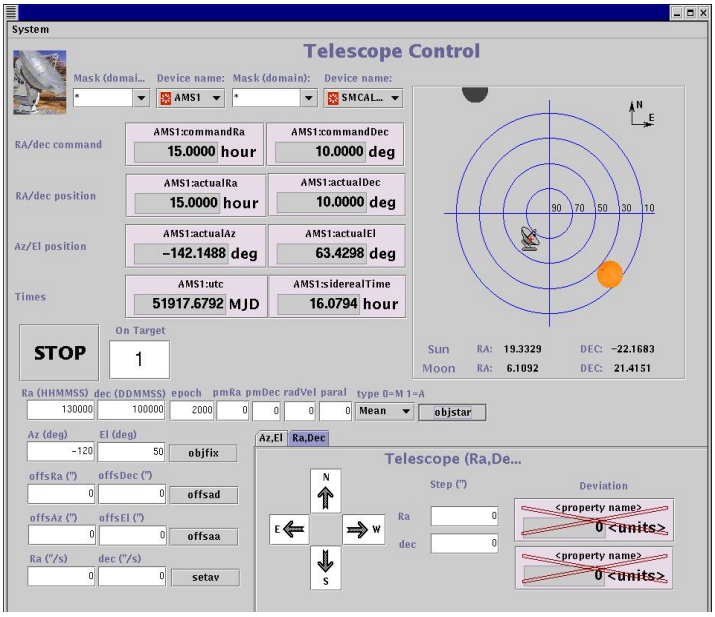

Fig . 6: Telescope Control GUI

However, beans can also be "invisible", having pure functionality without graphical representation. We have written a library of invisible Java beans, called Abeans for ACS beans that wrap Distributed Objects of ALMA. For each DO type there is one corresponding bean. A code generator automatically creates beans from the interface definition language (IDL) description of ALMA devices. A device bean encapsulates all remote calls from the client to a device server of the ACS, e.g. get/set, on/off. Thus the network is invisible to the user of Abeans. Tasks of an Abean include opening the connection and performing the function calls on remote objects; reporting and managing all errors/exceptions/timeouts, providing handles for asynchronous messages and the like.

The Abeans strive to present the final user with a view of ALMA as a collection of device beans, thereby reducing the problem of writing applications for the control system to the problem of familiarizing oneself with the ways to use Java beans and development tools. As much complexity as possible is hidden, but still accessible on demand. All interfaces that Abeans expose are type-safe (with no 'single function taking control string' methods) and enable one to construct simple yet powerful graphics application in a very short time.

In this way, the learning curve for a new programmer is smooth. For example, the ALMA control system uses CORBA for communication, but programmers writing the visual components or "views" are completely unaware of CORBA, since the invisible Abeans layer hides it. An example screenshot is shown in Fig.6.

Another type of Abeans applications are those that manage whole sets of devices at the same time, like generic device tables, alarm tables and snapshots. We have written several generic applications that can accept any device Bean, even if its type was not known at the time the applications were written:

- AlarmTable: registers as alarm-listener to all properties. Displays actual alarms; filters and mask alarms; (de)selects device/properties, etc.
- ObjectTable: displays all devices of a given class (type) in a table, one device per row. The columns are values of properties. ObjectTable also features trend-chart (values as function of time) and profile-chart (values as functions of position) of any combination of properties, multiple-device commands, save/load snapshot files, etc.
- Snapshot: displays all active devices in a Windows Explorer like tree, takes the actual set-values of properties for selected devices (by domain, type, etc.) and writes it to a text file or into an SQL database. The text file is in TAB-delimited format for easy import into spreadsheets. Existing snapshot files can be examined and selectively (by device type and domain) loaded.

7. ACS DEVELOPMENT STATUS

The development of ACS is driven by the needs of the teams developing higher-level software, and in particular the ALMA Control System.

7.1 ACS Release Policy

Our development cycle foresees one major release every year, with an intermediate, bug-fix release after six months. The complete ACS SW Development Plan, is available on the ACS Web Page [2].

ACS 0.0, released in September 2000 was essentially a concept demonstration prototype. With the support of some components taken from the VLT Control Software [13] it has been used to develop a prototype control system for the 12m Kitt Peak antenna. This was successfully tested in December 2000.

ACS 1.0, released in September 2001, is the first “production release”. It was followed in April 2002 by ACS 1.1. ACS 1.1 includes an essentially complete implementation of the BACI, MACI and Abeans packages, together with Error System, Logging System, the first elements of the C++ Application Framework and prototypes for many other packages. It still relies on some components of the VLT Common Software including, in particular, the VLT Real Time Database as configuration database engine.

The next major release, ACS 2.0, is foreseen for September 2002. The main objectives for this release are to become fully independent of the VLT Common Software and to provide a Configuration Database Engine based on an XML file hierarchy. The Time System and Notification Channel packages developed by the TICS team will be integrated into ACS. All existing components will be extended and stabilized.

We have started to work in parallel on the evolution of ACS to accommodate the needs of the ALMA Data Flow Subsystems (Archiving, Scheduling, Observation Tools, Pipeline, etc.). After ACS 2.0 the first prototypes for the Java Component-Container packages will be made available. Future releases will concentrate more on performance and scalability, for example with the implementation of federated Managers.

7.2 ACS Supported Platforms

ACS is supported for ALMA on Linux and VxWorks. An MS Windows version is running at the ANKA Synchrotron. A Solaris version is also used internally at ESO for testing purposes. Other platforms, like real time Linux, are being investigated, but the porting is expected to be very easy thanks to the ACE [14] operating system abstraction layer.

7.3 ACS Installations

ACS 1.1 is used in more than 10 sites worldwide for different projects mostly but not exclusively related to ALMA:

- For the development of TICS, the ALMA Test Interferometer Control System[4]. The first test antenna is being installed and tested at the VLA site.
- For the development of the APEX (Atacama Pathfinder Experiment), a radio telescope built by the Max Planck Institute for Radioastronomy in Germany, with the collaboration of ESO.
- For the development of the Japanese ALMA Antenna Prototype.
- It is in operation at the ANKA Synchrotron in Karlsruhe, (Germany), in its Windows NT porting.

Some ACS components, like the Abeans libraries are used by other projects as well and we are discussing the possibility of collaboration with other groups.

8. CONCLUSION

ACS has been developed keeping in mind the needs of a wide range of astronomical and accelerator control projects. It runs readily on many platforms and operating systems and is open source. The complete code is available under GNU Public License, is compiled with the standard GNU gcc compiler, and includes the sources of the underlying CORBA implementation, TAO[14], which is also open source. A part of the service client applications are written in Java, using a free Java ORB. We are therefore convinced that many other projects can use ACS. At the same time, we think that a wider user base can provide us with very valuable feedback.

ACKNOWLEDGEMENTS

The ACS project is managed by ESO in collaboration with JSI. This work is the result of many hours of discussions, test and development inside our groups and in the various ALMA centers at NRAO, IRAM and Bochum. We thank here all our colleagues for their important contributiond to the definition and implementation of ACS.

REFERENCES

1. ALMA Web page, http://www.mma.nrao.edu/
2. ACS web page and online documentation, http://www.eso.org/~gchiozzi/AlmaAcs
3. J.Schwarz, G.Raffi, “ALMA Software Architecture”, these proceedings
4. R.G.Marson, B.Glendenning, “ALMA Test Interferometer Control Software”, these proceedings
5. G.Raffi, G.Chiozzi, B.Glendenning, “The ALMA Common Software (ACS) as a basis for a distributed software development”, ADASS XI, Victoria, BC, Canada, Sep. 2001
6. M. Plesko, “Implementing Distributed Controlled Objects with CORBA”, PCaPAC99, KEK, Tsukuba, Jan. 1999
7. B. Jeram et al., “Distributed Components in Control”, ICALEPCS 1999, Trieste, Nov. 1999
8. G. Milcinski et al, “Experiences With Advanced CORBA Services”, ICALEPCS 2002, San Jose, CA, Nov. 2001
9. G. Tkacik et al., BACI specs, see [2]
10. K. Zagar et al., MACI specs, see [2]
11. G. Tkacik et al., “Java Beans of Accelerator Devices for Rapid Application Development”, PCaPAC99 workshop, KEK, Tskukuba, January 1999
12. CORBA Home Page at OMG: http://www.corba.org/
13. A.Wallander et al., “Status of VLTI control system: how to make an optical interferometer a data production facility”, these proceedings
14. ACE/TAO Home Page: http://www.cs.wustl.edu/~schmidt/TAO.html
15. K.Zagar et al., “The Control System Modeling Language”, ICALEPCS 2002, San Jose, CA, Nov. 2001
16. S.W.Ambler, *The Design of a Robust Persistence Layer for Relational Databases: an AmbySoft Inc. White Paper*, http://www.ambysoft.com/persistenceLayer.pdf
17. M.Völter, A.Schmid, E.Wolff, *Server Component Patterns*, Wiley, Summer 2002